|  |
| --- |
| Circle Language Spec: Commands |

## Commands Main Concepts

Commands are like executable objects. Commands can be actions, procedures and processes, that a computer could execute.

In other programming languages, commands are not objects, but entirely separate constructs, that you can not really exchange with objects. But in the new computer language, commands are all implemented as a special object, that can execute.

In other programming languages it is common for a process to be implemented as a procedure. But sometimes, when programming in other programming languages, you replace the procedure by an object, whose sole purpose is to execute a procedure. The object then hides complexity of the input and output better, and better hides the complexity of the procedure itself. Turning a procedure into an object also gives it more flexible linkage possibilities, for instance to be able to plug-in the process into an existing system. That is not easily possible, unless you turn a procedure into an object.

In the new computer language, every procedure actually *is* an object, that has the special property, that it is *executable*.

This means, that a command has all the complexity hiding and linkage possibilities, that an object has. A command for instance can also *inherit* the base procedure from another command.

This also means, that a command can be as well structured as an object, and it is no argument anymore, that using procedures is a less structured way of programming, than using objects, that represent processes. Every command is an object, that represents a process. It looks exactly like a command, but it has all the capabilities of an object.

Commands are *executable objects*.

You can use all the capabilities of objects inside a command. But a command has the special property, that it is *executable*.

### Diagram Notation

Commands are actions, that a computer can perform. Commands are executable objects.

Commands are displayed in a diagram as squares and diamond shapes.

|  |  |
| --- | --- |
|  |  |

Which shape to use when, is covered in other articles.

### Execute Once

Every command object can only be executed *once*. Each command object represents a single execution. An execution can be created and dormant until it is run. This gives you a chance to set the input of the command. After the command is run, the execution will stay created for long as it is referenced, so output can be read, until everybody is done with it. Only when the executable object is not referenced anymore, it gets destroyed. You cannot run the same command object twice: when you want to run the command again, you have to create a new command object with the same definition. You *can* *re*create an executable object, which means the old object is released, and a new object is created in its place. The reason why a command object is only executed once, is that this gives all the referrers a chance to read the execution’s output, whenever they want, without it being overwritten by new output. A command object stays created for as long as it is being referenced, so everybody can read the output of the command. The command object will only be destroyed when nothing refers to it anymore.

### Executable Commands

An *executable* command can be executed, unlike an *inactive* command, that stays dormant. Executable commands are also called *active commands*.

In a diagram every executable command symbol is displayed as a diamond shape:



A diamond shape is the symbol for *execution*.

A diamond shape is also called an *active command symbol*.

### Inactive Command

An inactive command object is dormant and never executes. An inactive command object can never be executed itself. It can only be used as a prototype.

In a diagram an inactive command is displayed as a square:



A square is the symbol for a command being dormant, *inactive*, as opposed to executing and active, like an *executable* command (see the article *Executable Command*).

### Command Definition

#### Concept

A command definition is a blue-print for another command.

There is a complete analogy between a command’s definition and the class of an object.

Anything that applies to classes, also applies to command definitions.

A command can select another command to function as its *prototype*. This prototype is called the *command definition*. The command definition describes the procedure of the command and how to link objects to the command. Commands with the same definition contain the same list of attributes, related items and related lists.

A command object will have the same structure as its definition, but not the same data. The values of the attributes can freely change for each individual command object. *Which* objects are referenced is also different for each individual command object. But initially the command object will be an exact replica of the definition. The definition’s attribute values and object references only function as a default.

A command, that is used as a definition, is usually not executable. Only individual calls to the command, that use the command as a definition, are executable. If a command definition is executable after all, then it is clearly mentioned, that it is an *active* command definition, because it is a special situation.

#### Diagram Notation

Any command object can serve as another command’s protype. So any command symbol can be the prototype for another object:

|  |  |
| --- | --- |
|  |  |

But usually, it is a square, that is another command’s definition:



When an *object* is drawn with a dashed line, then it is only used as a class:



This notation will *not* be copied to the concept of commands. This is because a command’s definition will usually be a square and a square will usually be a command’s definition, so using dashed lines for command definitions, would create an overload of dashed squares. So command symbols, that are only used as a definition, do not get a dashed notation.

### Command Call

#### Concept

A command definition can be called. A command call is the execution of a command definition.

A command call is an instantiation of a command definition. A command call is an individual object with a class reference to another command object. There is a complete analogy between calls to a definition and objects of a class, with the addition, that a call is *executable*.

A command call always selects another command to function as its definition. Any command object can function as another command’s definition.

Initially, the call will be an exact replica of its definition. However, the data of command definition only functions as a default configuration. The data of the call object can be altered before it runs and altered when it runs. What data of a command can be altered is covered later in the *Parameter* articles.

At first a command call is dormant, so that you get a chance to fill in its parameters. Then you can run the command call. A command call can only be run once (see the article *Execute Once*).

A command call selects its command definition with a *class* redirection, because the definition will be the command call’s *prototype*, and the call will always be its own individual object for which the command definition is a prototype.

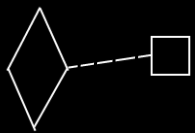
When a call is placed inside a parent command, the parent command, when run, automatically executes all the calls it contains. When a call is placed inside an *object*, the command can only be run manually.

#### Diagram Notation

A command call has a diamond shape, which stands for execution:



A call redirects its class to its definition. So a call is an active command symbol with a definition line:



Because the definition is the prototype of the call, and not the same object as the call, a dashed class line needs to be used to point out the definition of a call.

A call does not have to be placed inside another command. It can also be placed inside an object, in case of which somebody has to run the executable object manually.

### Command Reference

#### Concept

A command reference is a command, that *redirects* to another command object. This is done with *object* redirection.

A command reference is usually inactive. But a command reference can also be active.

The handy thing about command reference, is that it makes you able keep the operation to execute variable. The target of the command reference is variable. When you *call* a command reference, then the target of the command reference determines which command is called. So calling a command reference means calling a variable command definition.

A command reference represents the same object as the object pointed to. So the command reference’s contents are the exact same objects as that of the command object it points to.

A command reference can also redirect to yet another command reference, creating multiple command object redirections. The target of the last command reference determines the definition of the first command reference.

Both the command reference and its target can be either active or inactive.

An *active* reference to an *inactive* command can *not* be executed. The final target of object redirections is the object itself, and when it is inactive, the command object can *not* be executed, but only function as a prototype.

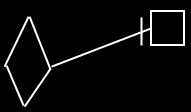
You can not execute an *active* command object through an *inactive* command reference either. But an active reference to an inactive reference to an active command *can* be executed.

#### Diagram Notation

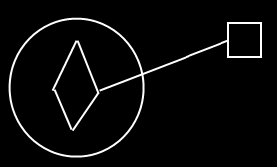
A command reference is displayed in a diagram as a diamond or square connected to another diamond or square with a solid line.



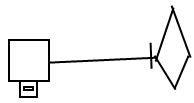
It does not matter whether it is squares or diamonds, because the only difference between a square and a diamond, is that a square can *not* be executed and a diamond *can*.



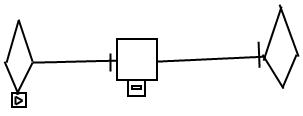
Direction of the line must be indicated with an access mark, unless the line is going outwards:



You can not execute an active command object through an inactive command reference.



But with an executable reference to an inactive reference to an executable command you *can* execute the command object again.



### Clause

#### Concept

A clause is a command defined within a command. A clause does not redirect its definition or object. A clause has a fixed logical residence inside another command. A clause can also reside inside *yet* *another* clause.

A clause can be *active* or *inactive*. If it is active, it is like a command call, executed when its parent command is executed. If a clause is *inactive*, then it is only executed when it is *called*.

A clause is always created as long as the parent command is created.

A clause does not redirect its definition, because then it would be a command call.

A clause does not redirect its object, because then it would be a command reference.

A clause is never situated inside an object, or it would not be a clause.

Clauses are like command definitions, therefore they can have parameters just like command definitions.

#### Diagram Notation

A clause is a command, defined within a command.

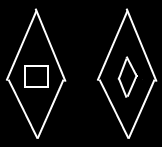
An active clause is shown in a diagram as a diamond placed inside a command symbol:



An *inactive* clause is shown in a diagram as a square inside a command symbol.

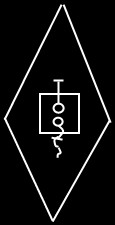


A clause can just as well be placed inside a diamond, instead of a square:



A clause will never redirect its definition or object to another command.

Clauses can have parameters, just like command definitions:



### Inactive Clause

A clause is a command defined within a command. The article *Clause* explains the general properties of clauses. An *inactive* clause is only executed when it is called. An inactive clause can be called and referenced like any other command. It is common to pass a reference to an inactive clause to an execution control command, such as a conditional execution or a loop (see the *Execution Control* articles). An inactive clause *is* a command definition, with a fixed logical residence inside another command.

#### In a Diagram

A clause is a command defined within a command.

An inactive clause is shown in a diagram as a square inside another command symbol.



A clause can just as well be placed inside a diamond, instead of a square:



A clause will never redirect its definition or object to another command.

### Active Clause

#### Concept

A clause is a command defined within a command. The article *Clause* explains the general properties of clauses. An *active* clause executes when its parent command is executed.

An active clause is actually more primitive than a command call, but command call was explained first, because it is more commonly used.

An active clause is like a call and a definition at the same time. It is analogus to an object that does not have a class. A *call* is more like an object that *does* have a class. Because an active clause has its own definition, its contents are totally arbitrary and definable by the author of the parent command, unlike calls, whose contents comply with the definition, that they call.

An active clause can not be redirected, because that would turn it into a command reference and not make it be an active clause anymore.

Active clauses can freely be used, to put a frame around a piece of code inside a command. In that case, the whole command will still do exactly the same thing.

#### Diagram Notation

The principle of active clauses is explained in the article *Active Clause*. The current article demonstrates its expression in a diagram.

A clause is a command defined within a command.

An active clause is shown in a diagram as a diamond placed inside a command symbol:

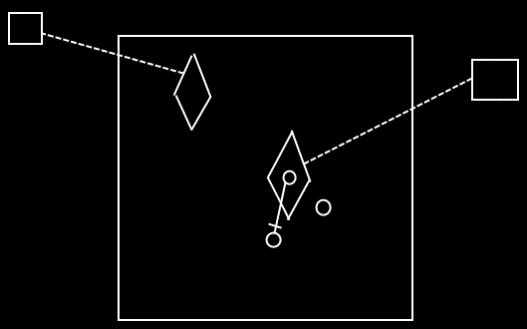


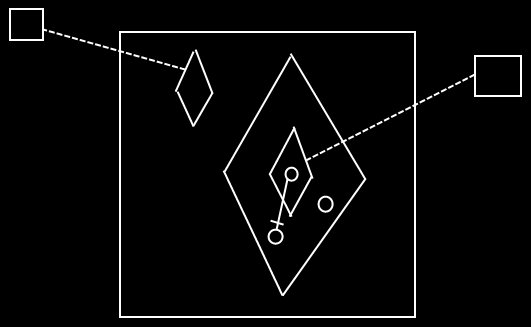
A clause can just as well be placed inside another diamond, instead of a square:



A clause will never redirect its definition or object to another command.

Active clauses can freely be used to put a frame around a piece of code inside a command. In that case the whole command will still do exactly the same thing:





### Commands Anywhere

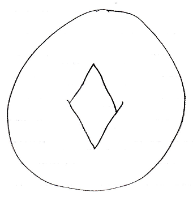
Commands are executable objects that can freely move around. You have to start thinking of an executable object more like an object, that just happens to be executable. You are going to have to see an execution much, much more like an executable object that can be situated anywhere and referenced from anywhere. You can put the executable object inside another object. You can reference an executable object. Yes, you can put an executable object inside a command definition. But you can also put an executable object inside a class.

You can run an execution inside an object, sort of like having an execution run inside a specific folder. You can also run the execution inside your user object and sort of take it along with you, whereever you go into the digital world. You can also run an execution globally on a site.

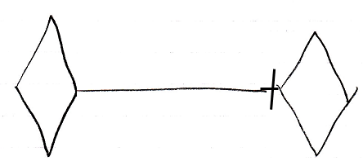
Executions can also be placed in a list. No problem. You can run each item in the list individually.

#### Diagram Notation

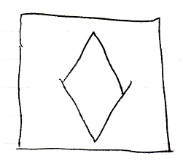
You can put the executable object inside another object:



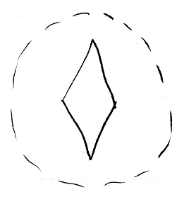
You can reference an executable object:



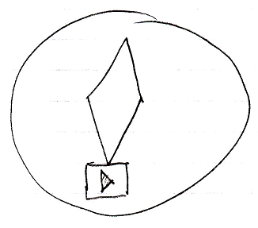
Yes, you can put an executable object inside a command definition:



But you can also put an executable object inside a class:

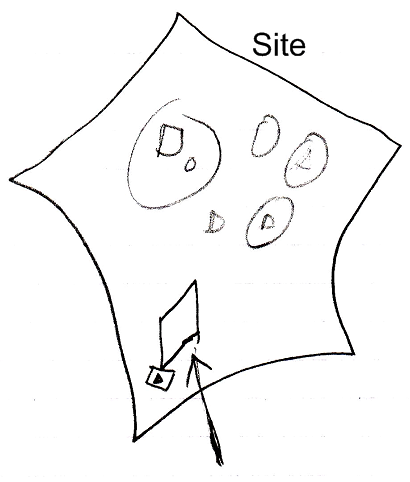


You can run an execution inside an object, sort of like having an execution run inside a specific folder:

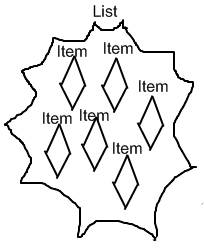


You can also run the execution inside your user object and sort of take it along with you, whereever you go into the digital world.

You can also run an execution globally on a site.



Executions can also be placed in a list. No problem. You can run each item in the list individually.



### Changing Inactive to Executable

Any part of a system can be changed, so an inactive command can be edited and changed to an executable command and back again.

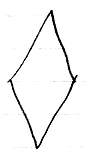
When you design a command definition, you might want to do it using an active command object, and test it once by running the active command definition. After that you can change it to an *inactive* command definition and give it the appropriate default values again. If you decide to again change the inactive command definition *back* to an active command, then you can not run it again, because the command object has already been run. Others might still have a reference to the execution to be able to read its output. If you want to run the command definition again, you are going to have to make a call to it instead.

If you change an inactive command definition to an executable command definition, then you can all of a sudden run it, which may overwrite the executable’s default output values.

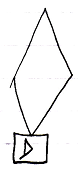
#### Diagram Notation

The article *Changing Inactive to Executable* has already explained these effect conceptually. The current article further clarifies it with diagrams.

When you design a command definition, you might want to do it using an active command object:



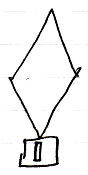
and test it once by running the active command definition:



After that you can change it to an inactive command definition and give it the appropriate default values.

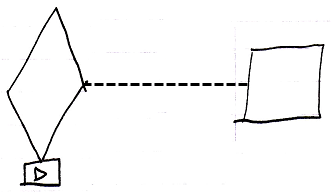


If you decide to change the inactive command definition back to an active command, then you can not run it again, because the command object has already run:

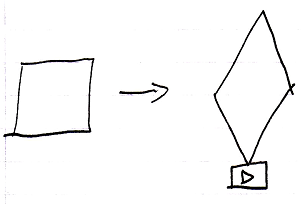


Others might still have a reference to it to be able to read its output.

If you want to run the command definition again, you are going to have to make a call to it instead:



If you change an inactive command definition to an executable command definition:



then you can all of a sudden run it, which may overwrite the executable’s default output values.

### Sub-Commands

Executions of commands can run inside a parent command. Sub-commands are *active* commands contained inside a *parent* command.

A parent command automatically executes its sub-commands. After a sub-command completes, the process returns to the parent command, which will then continue, executing the next sub-command.

Inside a command, usually just more commands are invoked.

There are only a few commands that do something other than execute other commands. Those are special commands, that perform a machine instruction: an operation that is executed by the CPU, the central processing unit of the computer. On top of those special commands, a few basic commands exist, like **If**’s and **For** loops, that control the flow of a program, making the next command to call dependent on a condition.

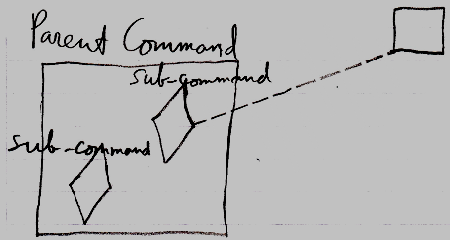
But basically, a command just calls more commands. Machine instructions, arithmetic operators, comparative and boolean algebra, assignments, and execution control statements such as **If** and **For**, are *all just commands*.

Apart from sub-commands, a command can also contain data.  
(Inactive clauses and inactive command references for instance are also considered data, and are not sub-commands, because they do not execute.)

#### Sub-Commands in a Diagram

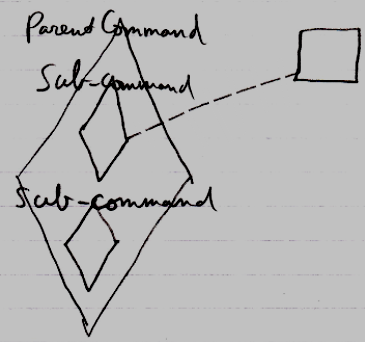
Sub-commands are active commands contained inside a parent command. You can encounter them inside any command symbol: both active and inactive commands symbols can contain sub-commands. The sub-commands are *active* commands: command calls, active clauses or active command references.

Below is a picture of two sub-commands inside an inactive command:



The **Parent Command** is an inactive command, because it is a square. In the diagram above, the **Parent Command** contains two sub-commands. The sub-commands are displayed as diamond shapes. One of the sub-commands is a call, because it has a dashed line going outside the **Parent Command**, tying the sub-command to its command definition. The other sub-command is an *active clause*, because it does not redirect its command definition.

Below is a picture of two sub-commands inside an active command:



It is the same picture as the other diagram, only now the **Parent Command** is an executable command symbol, not an inactive command.

### Command References Inside Commands

A command reference is considered data. A command reference in a parent command is considered data of the parent command, even when it is active, in case of which it *is* considered a sub-command but a special form of it. Even though it executes, it is still a reference to an executable object *elsewhere*. Only the *pointer* part is the data. Command references are *data* that can be *private* or *public*.

### Executables & Executions

An executable object stands for a potential execution. An executable object never stands for multiple executions, just one exectution or no execution at all.

So each execution is represented by its own individual executable object.

Only when an executable object is actually executed, it is called an execution. When an executable object is not executing yet, it is still only a *potential* execution.

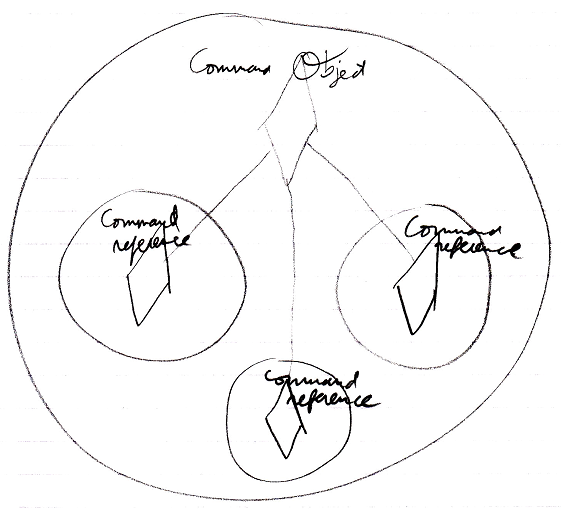
#### Diagram Notation

An executable object stands for a potential execution:



An executable object never stands for multiple executions, just one exectution or no execution at all.

The same command object can be displayed in the diagram multiple times. In that case, automatic containment will add a symbol to the diagram, that the multiple references to the same command object will converge to, so that a single symbol in the diagram is selected to represent the command object itself, while the other ones are just references.



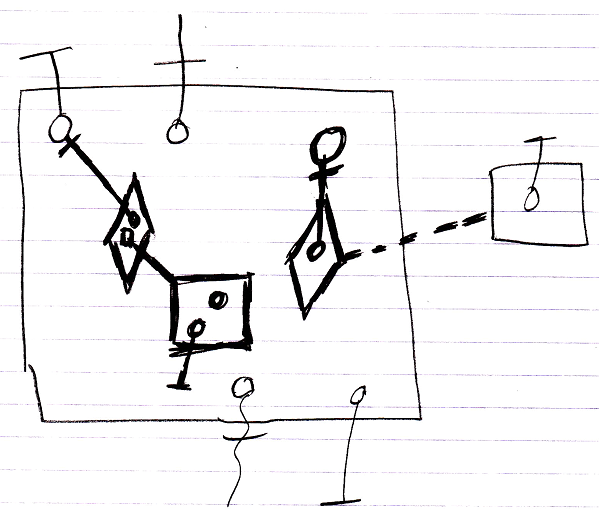
So it is not so, that each diamond in the diagram represents its own individual command execution. Each individual *command object* represents an individual command execution. Multiple symbols in the diagram can represent the same command object, but will converge into one symbol representing the object itself.

### Procedure

The implementation of a command is called a procedure. The implementation of a command are all the private contents of a command. So the private contents of a command are called the procedure of a command.

#### Diagram Notation

The term *procedure* is already defined in the article *Procedure*. The current article only shows the concept in a diagram.



The contents of the large square, drawn with thick lines, are the command’s *procedure*, because they are all private contents The objects contained inside the large square, that are drawn with normal lines, are the command’s parameters.

### Resolution When Not Allowed For Commands

When the command concept imposes a rule, that does not apply to normal objects, then you will seem to be able to break that rule. To keep direct conversion between an object and a command possible without any loss of structure, it seems to be allowed to break any of the extra rules imposed by commands. But in that case the command will always get the next best alternative behavior and a warning will be generated. This keeps conversion between objects and commands possible without any loss of structure.

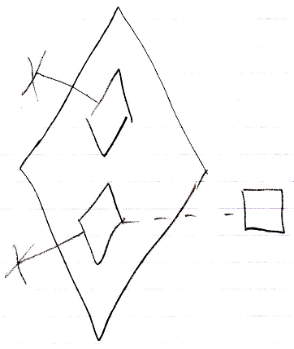
### Public Inactive Clause = Command Out Parameter

*Active* clauses, command calls and active command references in parent commands are always private, because you can not reference a sub-command (see the article *Sub-Commands Are Never Referenced*).

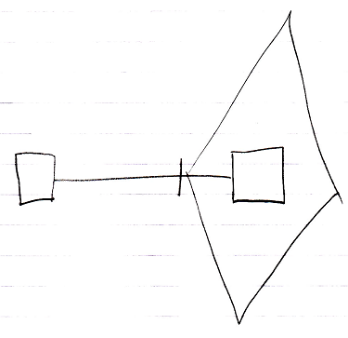
But *inactive* clauses *can* be referenced might be made public. If you make an inactive clause public, you will make it an **Object Out** parameter: an object produced by the command. A public inactive clause would be a command definition produced by another command. It would be an **Object Out** parameter, that is an executable object. That would work just fine.

#### Diagram Notation

Active clauses, command calls and active command references in parent commands are always private, because you can not reference a sub-command (see the article *Sub-Commands Are Never Referenced*).



But *inactive* clauses *can* be referenced might be made public.



If you make an inactive clause public, you will make it an **Object Out** parameter: an object produced by the command.

### Reading & Writing Parameters

Whether a parameter of a command can be read or written at all is access-controlled. This access control determines whether a parameter is for instance input or output.

Before a command is run you can mess about with the parameters all you want.

During the execution of a command you can not read or write anything.

After a command has executed, you can not change the parameters, just out of practical reasons, because it is more practical for the parameters to keep visualizing the state they were in after the command was run.

- Before execution you can read and write.

- During execution you can not read or write

- After execution you can only read

A command definition’s parameter values are public, so if you can reach the definition, you can basically change its parameter values, and mess about with them all you want. This needs to be access controlled, but how this should be done is not yet determined.

Details about parameters can be found in the *Parameters* articles.